

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

**Authoring and Using Generic Classes in JAVA  
Language Code**

Inventors:

**Makarand Gadre  
Pratap V. Lakshman**

ATTORNEY'S DOCKET NO. MS1-1597US

**EL996276256**

# **Authoring and Using Generic Classes in JAVA Language Code**

## **CROSS-REFERENCE TO RELATED APPLICATIONS**

The present application is related to co-pending U.S. patent application Ser. No. \_\_\_\_\_, Attorney Docket No. MS1-1596US, entitled "Compiling Source Code Using Generic Classes" by Makarand Gadre; which is filed concurrently herewith, assigned to the assignee of the present application, and incorporated herein by reference for all that it teaches and discloses.

## **TECHNICAL FIELD**

The subject matter disclosed herein relates generally to methods, devices and/or systems for compiling source code that uses generic classes.

## **BACKGROUND**

Frameworks include class libraries that provide software developers with a library of classes for developing, testing, using, and deploying software. Examples of two popular frameworks are the .NET Framework from Microsoft® Corporation of Redmond, Washington, and the JAVA™ language framework from Sun Microsystems, Inc. of Palo Alto, California. Generic classes (in C++ referred to as template classes; also referred to as generic types) may be provided by such frameworks.

Generic classes refer to classes, interfaces and methods that operate uniformly on values of different types. Generic classes can speed software development by packaging classes, methods, and data and making them applicable to multiple data types that are used frequently by developers. Generic classes are

1 useful because many common classes can be parameterized by the types of data  
2 being stored and manipulated – these are called generic class declarations.  
3 Similarly, many interfaces define contracts that can be parameterized by the types  
4 of data they handle – these are called generic interface declarations. Methods may  
5 also be parameterized by type in order to implement “generic algorithms”, and  
6 these are known as ‘generic methods’.

7  
8 A formal specification for a software language specifies standard syntax for  
9 the language. Formal specifications for C++ and other languages set forth generic  
10 class syntaxes that specify how generic classes are defined and declared (or,  
11 template class) in those languages; however, formal specifications for some  
12 languages, such as JAVA™ language, do not specify generic classes. Thus, generic  
13 classes that may be provided in frameworks, or other software packages, are not  
14 readily accessible by developers of JAVA™ language source code. For example,  
15 currently, JAVA™ language source code cannot use a generic class that may be  
16 provided by the .NET™ Framework. Thus, to take full advantage of a framework,  
17 developers need the capabilities for authoring, using, and compiling generic  
18 classes that may be provided by the framework.

## 19 20 **SUMMARY**

21 Implementations described herein provide methods and systems for  
22 compiling a generic class reference into an intermediate language executable by a  
23 runtime engine. The generic class may be referenced in source code written in a  
24 language for which use of generic classes is not formally specified.

1 An exemplary method includes writing JAVA<sup>TM</sup> language source code that  
2 includes a definition of a generic class, generating an instance of the generic class;  
3 and compiling the instance of the generic class into common intermediate  
4 language code executable by a runtime engine.

5  
6 An exemplary system receives input representing a generic class definition  
7 in a JAVA<sup>TM</sup> language, receives source code that references the generic class,  
8 compiles the source code with an instance of the generic class into common  
9 intermediate language code executable by a runtime engine.

10  
11 Additional features and advantages will be made apparent from the  
12 following detailed description of illustrative embodiments, which proceeds with  
13 reference to the accompanying figures.

### 14 15 **BRIEF DESCRIPTION OF THE DRAWINGS**

16 A more complete understanding of the various methods and arrangements  
17 described herein, and equivalents thereof, may be had by reference to the  
18 following detailed description when taken in conjunction with the accompanying  
19 drawings wherein:

20 Fig. 1 is a block diagram generally illustrating an exemplary computer  
21 system on which various exemplary technologies disclosed herein may be  
22 implemented.

23 Fig. 2 is a block diagram illustrating an exemplary framework, a compiled  
24 project and a runtime engine.

1        Fig. 3 is a block diagram illustrating an exemplary compiler operable to  
2 compile source code that references generic classes into project code executable  
3 by a runtime engine.

## 4 5                    **DETAILED DESCRIPTION**

6        Turning to the drawings, wherein like reference numerals refer to like  
7 elements, various methods and converters are illustrated as being implemented in a  
8 suitable computing environment. Although not required, the methods and  
9 converters will be described in the general context of computer-executable  
10 instructions, such as program modules, being executed by a personal computer.  
11 Generally, program modules include routines, programs, objects, components, data  
12 structures, etc. that perform particular tasks or implement particular abstract data  
13 types. Moreover, those skilled in the art will appreciate that the methods and  
14 converters may be practiced with other computer system configurations, including  
15 hand-held devices, multi-processor systems, microprocessor based or  
16 programmable consumer electronics, network PCs, minicomputers, mainframe  
17 computers, and the like. The methods and converters may also be practiced in  
18 distributed computing environments where tasks are performed by remote  
19 processing devices that are linked through a communications network. In a  
20 distributed computing environment, program modules may be located in both local  
21 and remote memory storage devices.

## 22 23                    **Overview**

24        Implementations described herein provide methods and systems for using  
25 generic classes in source code written in a language for which generic classes are

1 not formally specified. Generally, source code may be developed using a  
2 framework wherein generic classes are available. For example, generic classes  
3 associated with a framework capable of using multiple source codes and an  
4 intermediate language, can be referenced in a JAVA<sup>TM</sup> language. The source code  
5 is converted into an intermediate language source code. Metadata can be  
6 generated that describes any referenced generic classes.

7  
8 Thus, an implementation enables a Visual J# .NET<sup>TM</sup> (VJ#<sup>TM</sup>) Compiler to  
9 work with generic classes. In this regard, an improved VJ#<sup>TM</sup> compiler include  
10 support for generic types, including data structures, information, and algorithms  
11 that are processed and executed in connection with authoring and using generic  
12 types. In one implementation, the VJ#<sup>TM</sup> compiler applies an algorithm of parsing  
13 a variable or type declaration having references to generic classes, looking up  
14 reference assemblies and validating types with respect to the generic classes,  
15 utilizing data structures representing parsed and validated generics information,  
16 and traversing a generic tree representation to generate common intermediate  
17 language code.

## 18 19 **Exemplary Computing Environment**

20 Fig.1 illustrates an example of a suitable computing environment 120 with  
21 which the subsequently described exemplary methods, compilers, parsers, etc.,  
22 may be implemented.

23  
24 Exemplary computing environment 120 is only one example of a suitable  
25 computing environment and is not intended to suggest any limitation as to the

1 scope of use or functionality of the improved methods and arrangements described  
2 herein. Neither should computing environment 120 be interpreted as having any  
3 dependency or requirement relating to any one or combination of components  
4 illustrated in computing environment 120.

5  
6 The improved methods and arrangements herein are operational with  
7 numerous other general purpose or special purpose computing system  
8 environments or configurations. Examples of well known computing systems,  
9 environments, and/or configurations that may be suitable include, but are not  
10 limited to, personal computers, server computers, thin clients, thick clients, hand-  
11 held or laptop devices, multiprocessor systems, microprocessor-based systems, set  
12 top boxes, programmable consumer electronics, network PCs, minicomputers,  
13 mainframe computers, distributed computing environments that include any of the  
14 above systems or devices, and the like.

15  
16 As shown in Fig. 1, computing environment 120 includes a general-purpose  
17 computing device in the form of a computer 130. The components of computer  
18 130 may include one or more processors or processing units 132, a system  
19 memory 134, and a bus 136 that couples various system components including  
20 system memory 134 to processor 132.

21  
22 Bus 136 represents one or more of any of several types of bus structures,  
23 including a memory bus or memory controller, a peripheral bus, an accelerated  
24 graphics port, and a processor or local bus using any of a variety of bus  
25 architectures. By way of example, and not limitation, such architectures include

1 Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA)  
2 bus, Enhanced ISA (EISA) bus, Video Electronics Standards Association (VESA)  
3 local bus, and Peripheral Component Interconnects (PCI) bus also known as  
4 Mezzanine bus.

5  
6 Computer 130 typically includes a variety of computer readable media.  
7 Such media may be any available media that is accessible by computer 130, and it  
8 includes both volatile and non-volatile media, removable and non-removable  
9 media.

10  
11 In Fig. 1, system memory 134 includes computer readable media in the  
12 form of volatile memory, such as random access memory (RAM) 140, and/or non-  
13 volatile memory, such as read only memory (ROM) 138. A basic input/output  
14 system (BIOS) 142, containing the basic routines that help to transfer information  
15 between elements within computer 130, such as during start-up, is stored in ROM  
16 138. RAM 140 typically contains data and/or program modules that are  
17 immediately accessible to and/or presently being operated on by processor 132.

18  
19 Computer 130 may further include other removable/non-removable,  
20 volatile/non-volatile computer storage media. For example, Fig. 1 illustrates a  
21 hard disk drive 144 for reading from and writing to a non-removable, non-volatile  
22 magnetic media (not shown and typically called a "hard drive"), a magnetic disk  
23 drive 146 for reading from and writing to a removable, non-volatile magnetic disk  
24 148 (e.g., a "floppy disk"), and an optical disk drive 150 for reading from or  
25 writing to a removable, non-volatile optical disk 152 such as a CD-ROM, CD-R,



1 CD-RW, DVD-ROM, DVD-RAM or other optical media. Hard disk drive 144,  
2 magnetic disk drive 146 and optical disk drive 150 are each connected to bus 136  
3 by one or more interfaces 154.

4  
5 The drives and associated computer-readable media provide nonvolatile  
6 storage of computer readable instructions, data structures, program modules, and  
7 other data for computer 130. Although the exemplary environment described  
8 herein employs a hard disk, a removable magnetic disk 148 and a removable  
9 optical disk 152, it should be appreciated by those skilled in the art that other types  
10 of computer readable media which can store data that is accessible by a computer,  
11 such as magnetic cassettes, flash memory cards, digital video disks, random access  
12 memories (RAMs), read only memories (ROM), and the like, may also be used in  
13 the exemplary operating environment.

14  
15 A number of program modules may be stored on the hard disk, magnetic  
16 disk 148, optical disk 152, ROM 138, or RAM 140, including, e.g., an operating  
17 system 158, one or more application programs 160, other program modules 162,  
18 and program data 164.

19  
20 The improved methods and arrangements described herein may be  
21 implemented within operating system 158, one or more application programs 160,  
22 other program modules 162, and/or program data 164.

23  
24 A user may provide commands and information into computer 130 through  
25 input devices such as keyboard 166 and pointing device 168 (such as a "mouse").

1 Other input devices (not shown) may include a microphone, joystick, game pad,  
2 satellite dish, serial port, scanner, camera, etc. These and other input devices are  
3 connected to the processing unit 132 through a user input interface 170 that is  
4 coupled to bus 136, but may be connected by other interface and bus structures,  
5 such as a parallel port, game port, or a universal serial bus (USB).  
6

7 A monitor 172 or other type of display device is also connected to bus 136  
8 via an interface, such as a video adapter 174. In addition to monitor 172, personal  
9 computers typically include other peripheral output devices (not shown), such as  
10 speakers and printers, which may be connected through output peripheral interface  
11 175.  
12

13 Logical connections shown in Fig. 1 are a local area network (LAN) 177  
14 and a general wide area network (WAN) 179. The LAN 177 and/or the WAN 179  
15 can be wired networks, wireless networks, or any combination of wired or wireless  
16 networks. Such networking environments are commonplace in offices, enterprise-  
17 wide computer networks, intranets, and the Internet.  
18

19 When used in a LAN networking environment, computer 130 is connected  
20 to LAN 177 via network interface or adapter 186. When used in a WAN  
21 networking environment, the computer typically includes a modem 178 or other  
22 means for establishing communications over WAN 179. Modem 178, which may  
23 be internal or external, may be connected to system bus 136 via the user input  
24 interface 170 or other appropriate mechanism.  
25

1 Depicted in Fig. 1, is a specific implementation of a WAN via the Internet.  
2 Here, computer 130 employs modem 178 to establish communications with at  
3 least one remote computer 182 via the Internet 180.

4  
5 In a networked environment, program modules depicted relative to  
6 computer 130, or portions thereof, may be stored in a remote memory storage  
7 device. Thus, e.g., as depicted in Fig. 1, remote application programs 189 may  
8 reside on a memory device of remote computer 182. It will be appreciated that the  
9 network connections shown and described are exemplary and other means of  
10 establishing a communications link between the computers may be used.

### 11 12 **Exemplary Framework for Authoring, Using, and Compiling Generic Classes**

13 Fig. 2 shows an exemplary framework 200 and a compiled project 202  
14 targeted for execution on a runtime engine (RE) 204. In object-oriented  
15 programming, the terms “Virtual Machine” (VM) and “Runtime Engine” (RE)  
16 have recently become associated with software that executes code on a processor  
17 or a hardware platform. The RE 204 is operable to translate common intermediate  
18 language code into microprocessor-specific binary that is executable by a  
19 computer. In the description presented herein, the term “RE” includes VM. A RE  
20 is often associated with a larger system (e.g., IDE, framework, etc.) that allows a  
21 programmer to develop an application.

22  
23 For a programmer, the application development process usually involves  
24 selecting a framework, coding in an object-oriented programming language  
25 (OOPL) associated with that framework to produce a source code, and compiling

1 the source code using a compiler associated with the framework. In Fig. 2, the  
2 framework 200 includes a code editor 206 for authoring (i.e., writing and/or  
3 editing) project source code 208, project resources 210 (e.g., libraries, utilities,  
4 etc.) and a compiler 212 for compiling the project source code 208. The  
5 programmer may elect to save project source code and/or project resources in a  
6 project file and/or a solution file, which may contain more than one project file. If  
7 a programmer elects to compile project code and/or project resources, then the  
8 resulting compiled code, and other information if required, is then typically made  
9 available to users, e.g., as a compiled project, a solution, an executable file, an  
10 assembly, etc.

11  
12 The project resources 210 include class libraries 214 and other resources  
13 216 (e.g., utilities, etc.). The class libraries 214 have definitions for classes that  
14 may be used and/or authored by a developer. The classes contained in class  
15 libraries 214 may have associated tokens for ease of referencing and compiling the  
16 classes. For example, each class in the class libraries 214 can have a numerical  
17 token that identifies the class.

18  
19 One or more of the class definitions in the class libraries 214 correspond to  
20 generic classes (also called generic types) (e.g., generic classes 314, Fig. 3). The  
21 term “generic class” refers to classes, interfaces and methods that operate  
22 uniformly on instances of different types and/or classes. By way of example, and  
23 not limitation, a “Queue<Type>” class can be a generic class, wherein “Type” may  
24 be declared as any of multiple allowable types or classes. The class library  
25

1 definition of a generic class defines which types are allowable for the generic class  
2 as well as the methods applicable to an instance of a generic class.  
3

4 One or more standard generic classes may be provided by the framework  
5 200. For example, a recently developed framework called the .NET<sup>TM</sup> Framework  
6 (Microsoft Corporation, Redmond, Washington) comes with a generic "Queue  
7 <Type>" class, a "Dictionary<Type1, Type2>" class, a "Stack<Type>" class, and  
8 others. In addition, implementations of authoring methods and systems described  
9 herein enable a developer define generic classes and make them available in the  
10 class libraries 214 for use by the project code 208.

11  
12 Precompiled data 218 shown in Fig. 2 includes any data created and/or used  
13 by the compiler 212 to generate the compiled project 202. As is discussed in  
14 further detail below, precompiled data 212 may include a parse tree 312 (Fig. 3), a  
15 tokenized parse tree 316 (Fig. 3), and a validated tokenized parse tree 318 (Fig. 3).  
16 Precompiled data includes various data structures and other information that are  
17 intermediate between the source code 208 and the compiled project 202. Fig. 3  
18 describes exemplary data and information in the precompiled data 218 and how  
19 the compiler 212 uses the precompiled data to create the compiled project 202.  
20

21 Fig. 2 shows a compiled project 202 generated by the compiler 212, which  
22 includes portable code 220, metadata 222, and other data 224 (e.g., headers, native  
23 image data, custom image data, etc.) that may be necessary for proper execution of  
24 the portable code 220. The other data 224 may pertain to project resources 210 or  
25 other resources. The compiled project 202 is typically available as one or more

1 files capable of distribution over a network. For example, the .NET™ framework  
2 can produce a compiled project as a portable executable file containing  
3 intermediate language code (IL code) and metadata, which is suitable for  
4 distribution over the Internet and execution using the .NET™ RE. In the .NET™  
5 environment, the compiled project 214 may be referred to as an assembly. Of  
6 course, one or more separate code files and one or more separate data files may be  
7 contained within a project file or a compiled project file. Upon receipt of the  
8 requisite file or files, a user can execute an embedded application or applications  
9 on a RE associated with the selected framework. Fig. 2 shows the RE 204  
10 associated with the framework 200.

11  
12 Traditional frameworks, such as the JAVA™ language framework (Sun  
13 Microsystems, Inc., Palo Alto, California), were developed initially for use with a  
14 single OOPL (i.e., monolithic at the programming language level); however, the  
15 .NET™ Framework allows programmers to code in a variety of OOPLs (e.g.,  
16 VISUAL BASIC®, C++, Visual C# .NET™, JScript, Visual J# .NET™, etc.).  
17 This multi-OOPL or multi-source code framework is centered on a single  
18 compiled intermediate language having a virtual object system (VOS).

19  
20 The intermediate language (IL) generated by the .NET™ Framework is  
21 often referred to as a “language-neutral” intermediate language because the IL  
22 may be generated from software written in multiple source code languages. The  
23 compiler 212 in a .NET™ Framework compiles all source code to a common IL,  
24 irrespective of the source code language.  
25

1 In contrast to the .NET™ Framework, other frameworks, such as the  
2 JAVA™ language framework, do not allow programmers to code in a variety of  
3 OOPLs. For example, the JAVA™ language framework requires that all source  
4 code be in the JAVA™ language. The JAVA™ language framework compiles the  
5 JAVA™ language source code into bytecodes, which are non-language-neutral.  
6 Thus, in the JAVA™ language framework there cannot be bytecodes generated  
7 from multiple OOPLs.

8  
9 While the aforementioned .NET™ Framework exhibits programming  
10 language or source code interoperability, a need exists for methods, devices and/or  
11 systems that allow authorship, use, and compilation of generic classes in a JAVA™  
12 language project, solution, or source code. For example, a developer may want to  
13 declare a predefined generic class in source code written in the JAVA™ language,  
14 whereby the declared generic class is compiled into portable code. As further  
15 described herein, exemplary methods, devices, and/or systems can facilitate  
16 authoring, using, and compiling JAVA™ language source code in the .NET™  
17 Framework.

### 18 19 **Implementing Generic .NET™ Classes in a JAVA™ Language**

20 With particular regard to the code editor 206, a user may author the project  
21 source code 204 in a number of source code languages, including JAVA™, VJ++,  
22 Visual J# .NET™, or other JAVA™ languages. As used herein, the term “JAVA™  
23 language” refers to any source code language that is based on a formal JAVA™  
24 language specification, such as, but not limited to, the JAVA™ Development Kit  
25 (JDK™) 1.1.4. Although formal JAVA™ specifications do not specify generic

1 classes, exemplary implementations described herein provide ways for generic  
2 classes to be authored, used and compiled in a JAVA™ language source code.

3  
4 Implementations of methods and systems described herein enable authoring  
5 generic classes in JAVA™ language source code for use by JAVA™ language  
6 and/or software programs in other languages. In particular, these implementations  
7 provide for authoring and using generic classes whereby instances of such generic  
8 classes can be compiled into a common intermediate language (CIL) and executed  
9 by a runtime engine, such as the runtime engine 204. A generic class may be  
10 authored by defining the generic class such that methods and data of the generic  
11 class are uniformly applicable to multiple different classes. In addition, such  
12 generic classes authored in JAVA™ language may be used (e.g., declared,  
13 referenced, etc.) by software programs written in other languages, such as C++  
14 and Visual C# .NET™.

15  
16 In a particular implementation, angular brackets are used in JAVA™  
17 language source code to identify classes associated with a generic class. Between  
18 the angular brackets, at least one unconstrained type or class is specified. The  
19 following examples illustrate how a developer may author a generic class in  
20 JAVA™ language source code.

21  
22 Example 1:

```
23     public class MyGenericClass<X>  
24     {  
25         public MyGenericClass()  
26         {  
27             // constructor  
28         }  
29     }
```



```

1      public void Set(X xvar)
2      {
3          // code that may change state of this class
4      }
5
6      public X ReturnResult()
7      {
8          X xvar;
9          // code that may change xvar
10         return xvar;
11     }
12 }

```

13  
14 Example 1 illustrates a generic class definition in JAVA™ language source  
15 code in which the type argument, identified by ‘X’, can be of any class. The ‘X’  
16 class is called an unconstrained type because it can be of any class. The generic  
17 class can be instantiated by providing a value for the type argument. In so doing, a  
18 ‘constructed type’ is created.

19  
20 A second example of a generic class definition in JAVA™ language source  
21 code is shown in example 2 shown below

22 Example 2:

```

23      public class MyGenericClass<X implements Comparable>
24      {
25          public MyGenericClass()
26          {
27              // constructor
28          }
29
30          public void Set(X xvar)
31          {
32              // code that may change state of this class
33          }
34
35          public X ReturnResult()
36          {
37              X xvar;
38              // code that may change xvar

```

```
        return xvar;
    }
}
```

Example 2 illustrates how for certain generic classes each type-parameter may be qualified by an explicit-type-parameter-constraint. The specification of an explicit constraint is optional. If given, a constraint is a reference-type that specifies a minimal “type-bound” that every instantiation of the type parameter must support (for example, the constraint may be that the type parameter must implement a certain interface, inherit from a certain class, or provide a default constructor). In Example 2 above, the generic class can be instantiated by providing a value for the type argument, identified by ‘X’; the value provided must be of a class that implements the IComparable interface.

The foregoing examples illustrate how a developer may author generic classes in the JAVA™ language using the code editor 206. Such authored generic classes can be included in the generic classes of the class libraries 214. Other generic classes and types may be provided in the class libraries 214. As discussed earlier, such generic classes, whether or not they are authored in JAVA™ language, may be used by JAVA™ language programs and/or other non-JAVA™ language programs.

In a .NET™ Framework implementation, the generic classes (i.e., types), parameters, non-generic classes, and instantiated generic classes are defined by various code sections, such as .NET Assemblies, .NET Class Libraries, and User

Code. A .NET™ Assembly is a collection of classes in MSIL form (e.g., classes available in .NET™ Frameworks). Table 1 illustrates an exemplary arrangement.

**Table 1**

Description	Defined By
.NET Generic Type	.NET Class Libraries, .NET Assembly
Formal Parameter Type to a .NET Generic Type	.NET Class Libraries, .NET Assembly
Type Parameter of Generic Type to be instantiated	User Code
Non Generic Type	.NET Class Libraries, .NET Assembly, User Code
Constructed Type	User Code

Thus, a .NET Class Library and/or a .NET Assembly contain definitions of generic classes, and specify the formal parameter types/classes that can be passed to a generic class. User code, such as project code 208 and user-authored class libraries, specifies any constructed types (i.e., instantiated generic classes).

Generic classes that have been defined and stored in the class libraries 214 can be used by developers, even in source code written in languages for which the use of generic classes has not been formally specified, through implementations described herein. For example, a developer can declare, or otherwise reference, a generic class in JAVA™ language source code. In the .NET™ framework, a developer can create JAVA™ language source code using Visual J# .NET™ that includes declarations of instances of pre-defined generic classes. As is discussed in further detail below, the compiler 212 is operable to compile declared instances

of generic classes into portable code 220 in languages that do not formally specify use of generic classes.

With regard to using generic classes, a developer specifies in the source code any unconstrained types or classes defined in the generic class definition. As discussed above, when source code declares an instance of a generic class with an allowable unconstrained type, the instance of the generic class is referred to as a constructed class or type. A constructed class is a species of the generic class. For example, if a generic class Queue, is defined as 'Queue<X>,' wherein class 'X' is unconstrained, a declaration of 'Queue<int>' is referred to as a constructed class.

A developer specifies a constructed class of the desired generic class, and then uses the constructed class much like other classes. The developer can declare an instance of the constructed class, reference the instance of the constructed class, apply operations or methods to the instance of the constructed class, and the like.

Examples of declared generic types are shown below in Table 2, in which parameter 'X' refers to an unconstrained type:

**Table 2**

Declared Generic Type	Instantiated Type
Queue<X>	Queue <int> abc = new Queue <int>;
	Queue <System.String> abc = new Queue<System.String>;
	Queue <Queue <System.String> > = new Queue <Queue <System.String> >;
Lookup<int,X>	Lookup<int, Object> lu = new Lookup<int, Object>;
	Lookup <int, Queue<String> > = new Lookup<int, Queue<String> >; // Nested Generic Type

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25	<pre> class STR extends String ... SLookup&lt;String, Object&gt; slu = new SLookup&lt;String, Object&gt;; SLookup&lt;STR, Object&gt; slu2 = new SLookup&lt;STR, Object&gt;; // The next line would be error because // System.IntPtr is not an instanceof(String); SLookup&lt;System.IntPtr, Object&gt; = new SLookup&lt;System.IntPtr, Object&gt;; </pre>
---	--

Some generic classes may allow for nesting of classes. Nested classes refer to classes within classes. For example, a constructed class of the generic class 'Queue<X>' may be 'Queue<Queue<int>>,' wherein 'int' is a nested class; i.e., 'int' is nested in the inner 'Queue<>' generic class. In the foregoing example, because 'X' is unconstrained, generic classes can be nested at any number of levels. Thus, a constructed class takes the general form 'GC<GC<GC<...>>,' where 'GC' refers to the generic class. In a .NET™ implementation, nested classes may be used in JAVA™ language source code, and source code of other languages that may not formally specify use of generic classes.

Existing JAVA™ language source code can be easily adapted to use resources, such as generic classes, which may be provided by a framework or other software development package. In a framework environment, the adapted JAVA™ language source code can be compiled for execution by a runtime engine. A developer can modify existing source code to include references to generic classes. The developer simply needs to identify a generic class that is available from the framework or other software development package, and specify the class (or classes) that are unconstrained parameters for the generic class using the proper syntax. The developer creates a constructed class by declaring a generic

1 class specifying the unconstrained class (or classes) to be used. An instance of the  
2 constructed class can then be declared and used.

3  
4 For example, a JAVA<sup>TM</sup> language source code developer may want to port  
5 existing JAVA<sup>TM</sup> language code to the .NET<sup>TM</sup> Framework and use the generic  
6 classes provided by .NET<sup>TM</sup>. The existing JAVA<sup>TM</sup> language code may have been  
7 written in standard JAVA<sup>TM</sup> language or in a variation of JAVA<sup>TM</sup> language such as  
8 Visual J#<sup>TM</sup>, Jscript, or J++. Regardless of the original JAVA<sup>TM</sup> language used,  
9 Visual J# .NET<sup>TM</sup> in the .NET<sup>TM</sup> Framework enables a developer to port the  
10 existing JAVA<sup>TM</sup> language code to the .NET<sup>TM</sup> Framework and use the generic  
11 classes of the .NET<sup>TM</sup> Framework.

### 12 13 **Generating Executable Code From Source Code Using Generic Classes**

14 Compiling source code that uses generic classes involves generating a  
15 compiled project representative of the source code. The compiled project is  
16 readily executable by a microprocessor, using a runtime engine. The compiled  
17 project may also be portable to various platforms, hardware, etc. A common  
18 intermediate language (CIL) can facilitate portability of the compiled project.

19  
20 Thus, one implementation of portable code 218 includes a common  
21 intermediate language (CIL), such as Microsoft® Intermediate Language (MSIL)  
22 code. MSIL defines a virtual instruction set. The MSIL is typically translated by  
23 the runtime engine 222 into lower-level instructions executable by a  
24 microprocessor. The MSIL is portable by virtue of the fact that the runtime engine  
25 222 is microprocessor or platform aware. A particular implementation of the

1 framework 200 includes Visual J# .NET™. Visual J# .NET™ includes an editor  
2 for writing and editing source code using JAVA™ language syntax, and a compiler  
3 for compiling the JAVA™ language source code into Microsoft ® Intermediate  
4 Language (MSIL).

5  
6 Fig. 3 is a block diagram illustrating an exemplary compiler 212  
7 performing operation with respect to project code 208 to generate portable code  
8 220 executable by a runtime engine. The compiler 212 includes a parser 302,  
9 lexical analyzer (lexer) 304, common intermediate language (CIL) importer 306,  
10 semantic analyzer 308, and code generator 310.

11  
12 The parser 302 receives the project source code 208 or other input and  
13 generates lexemes based on the source code 208. A lexeme is a minimal lexical  
14 unit of a computing language, such as a keyword, identifier, literal, punctuation,  
15 and the like, that is recognized by the lexer 304. Typically, the stream of  
16 characters making up the source program 208 is read by the parser 302, one at a  
17 time, and grouped into lexemes, which are passed to the lexer 304.

18  
19 In one implementation, the parser 302 reads JAVA™ language source code  
20 from the project code 208, which includes references to generic classes 314. The  
21 parser 302 divides a reference to a generic class into the generic class name, and  
22 one or more associated classes, which may be constrained or unconstrained. For  
23 example, if 'Queue<X>' is a generic class, a declaration 'Queue<int>' may be  
24 divided into lexemes 'Queue' and 'int'.  
25

1       The lexer 304 analyzes the syntax of the lexemes generated by the parser  
2 302 with respect to a formal computing grammar. The lexer 304 resolves the  
3 lexemes into identifiable parts before translation into lower level machine code.  
4 The lexer 304 may also check to see that all input has been provided that is  
5 necessary. During compilation, the lexer 304 issues an error if the lexemes cannot  
6 be resolved to identifiable parts defined in the formal computing grammar.

7  
8       In one implementation, the output of the lexer 304 is a parse tree 312. The  
9 parse tree 312 is a representation of the source code 208 in which types referenced  
10 in the project code 208 are separated in preparation for code generation. The parse  
11 tree 312 may be a hierarchical, or tree, structure, in which parameters of generic  
12 class declarations are listed under the generic class. Nested classes of a generic  
13 class reference are presented at lower branches under the generic class. For  
14 example, a line of Visual J# .NET™ (a JAVA™ language) source code  
15 MyGenericClasses.LookupTable<long, MyGenericClasses.Queue<String> > may  
16 be represented in the parse tree 312 as follows:

```
17  
18       CType => MyGenericClasses.LookupTable  
19       ClassTree =>  
20           CType => long  
21           ClassTree => null  
22           CType => MyGenericClasses.Queue  
23           ClassTree =>  
24               CType => String  
25               ClassTree = null,
```

23       wherein 'LookupTable' is a generic class, having two parameters, in which  
24 the second parameter is unconstrained as to type. In the above example, the  
25 second parameter of the 'LookupTable' is 'Queue,' which is a generic class



1 having a nested class of 'String.' The parser interacts with the CIL importer 306  
2 to validate direct references to the generic classes based on metadata that describes  
3 the generic classes.

4  
5 In an exemplary implementation, the lexer 304 constructs variables of type  
6 CClass\_Type from the project code 208. CClassType is a subclass of Class  
7 CType. In this implementation, the parser 302 fills in recursive (i.e., nested)  
8 CClass\_Types for generic classes. Later, the lexer 304 traverses the tree while  
9 validating each CType and obtaining an associated CClass object reference. When  
10 the CClass object reference is created, the CIL importer 306 is called, which allots  
11 a token to the CClass object. CClass\_Type, CClass and CClass\_Info objects are  
12 kept unique for the duration of the compiler session.

13  
14 Thus, the CIL importer 306 generates a tokenized parse tree 316 based on  
15 the parse tree 312 and generic class definitions in the generic classes 314. The  
16 generic classes 314 may be obtained from class libraries (e.g., class libraries 208,  
17 Fig. 2) or other compiled projects (e.g., assemblies in .NET™). In the tokenized  
18 parse tree 312, the types are represented as tokens that refer to defined types. For  
19 example, a constructed class 'Queue<int, string>' may be represented in the parse  
20 tree 312 as follows:

```
21     TokenCurrent  
22         Token1  
23         Token2,
```

23 wherein "TokenCurrent" is a token associated with generic class 'Queue,' Token1  
24 is a token associated with class 'int', and Token2 is a token associated with class  
25 'string.'

1  
2 A particular implementation of the CIL importer 306 also generates  
3 metadata related to the classes referenced in the project code 208. The CIL  
4 importer 306 gathers metadata from class definitions and populates the tokenized  
5 parse tree 316 with the metadata.

6  
7 In a .NET™ implementation of the CIL importer 306, the CIL importer  
8 creates Microsoft® Intermediate Language (MSIL) assembly tokens using native  
9 .NET™ Metadata Application Programming Interfaces (APIs). The CIL importer  
10 306 uses the CClassType data created by the parser 304 to construct data of type  
11 CClass. CClass variables store ClassInfo, which include metadata descriptive of  
12 the class. The CIL importer 306 stores the MSIL assembly tokens in data of type  
13 CClassInfo. Every CClass\_Type has a field to hold the CClass and vice versa.

14  
15 class CType\_List : public std::list<const CType\*>  
16 {  
17 ...  
18 }

19 CClass\_Type holds a reference to CClass  
20 class CClass\_Type : CType

21 {  
22 ...  
23 CType\_List \*m\_pCtypeList;  
24 CClass \*pCClass;  
25 ...  
26 }

27 // CClass holds a reference to CClassInfo and a reference  
28 to CClass\_Type

29 class CClass  
30 {  
31 ...  
32 CClass\_Type \*pCClassType;  
33 CClass\_Info \*pCClass\_Info;  
34 ...  
35 }

```

1         CClass_Info
2         {
3             ...
4             unsigned int uAssemblbyToken;
5             ...
6         }

```

4 Metadata describes the types and classes in the portable code. Exemplary  
 5 metadata include: a name of the class; visibility information indicating the  
 6 visibility of the class; inheritance information indicating a class from which the  
 7 class derives; interface information indicating one or more interfaces implemented  
 8 by the class; method information indicating one or more methods implemented by  
 9 the class; properties information indicating identifying at least one property  
 10 exposed by the class; and events information indicating at least one event the class  
 11 provides.

13 The semantic analyzer 308 performs semantic analysis on the tokenized  
 14 parse tree 314. Semantic analysis involves traversing the tokenized parse tree 314  
 15 and validating types and operations with respect to the generic classes represented  
 16 in the parse tree. For example, the semantic analyzer 308 validates assignments  
 17 and casts with 'instanceof checks' to ensure that objects of generic classes are not  
 18 assigned to an invalid type. If invalid types or operations are identified by the  
 19 semantic analyzer 308, an error is generated during compile time. If no errors are  
 20 identified, the semantic analyzer 308 generates a validated tokenized parse tree  
 21 318.

23 The code generator 310 generates the compiled project 214 based on the  
 24 validated tokenized parse tree 318 and the generic classes 314. Code generator  
 25 310 converts the parsed and type checked tokens of the validated tokenized tree

1 318 into common intermediate language (CIL) code. The code generator 310  
2 traverses the validated tokenized parse tree 318 gathering tokens. When the code  
3 generator has enough tokens to create a line of CIL code, the corresponding CIL  
4 code is appended to the portable code 216.

5  
6 The code generator 214 creates the metadata 218 based on metadata in the  
7 validated tokenized parse tree 318. The metadata 218 may be stored with the  
8 project code 216 so that the compiled project 214 can be easily transported from  
9 one platform to another platform. In addition, the metadata 218 can enable  
10 another application program and/or developers to use the project code 216.

11  
12 Although some exemplary methods and systems have been illustrated in the  
13 accompanying Drawings and described in the foregoing Detailed Description, it  
14 will be understood that the methods and systems are not limited to the exemplary  
15 embodiments disclosed, but are capable of numerous rearrangements,  
16 modifications and substitutions without departing from the spirit set forth and  
17 defined by the following claims.